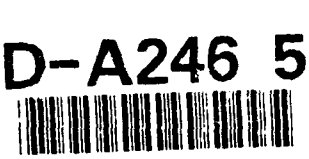
REPORT DO

AD-A246 509



Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 12 Jun 1991 to 01 Jun 1993

4. TITLE AND SUBTITLE
TeleSoft, IB<M Ada 370, Version 1.2.0 (with optimization) IBM 4381, MVS/ESA Rel. 3.1 (Opt) (Host & Target), 910612W1.11167

5. FUNDING NUMBERS
(2)

6. AUTHOR(S)
Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB, Dayton, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-474-0292

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for pubiic release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
TeleSoft, IB<M Ada 370, Version 1.2.0 (with optimization) IBM 4381, Wright-Patterson AFB, MVS/ESA Rel. 3.1 (Opt) (Host & Target), ACVC 1.11.

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

AVF Control Number: AVF-VSR-474-0292
4 February 1992
91-04-24-TEL

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910612W1.11167
TeleSoft
IBM Ada/370, Version 1.2.0 (with optimization)
IBM 4381, MVS/ESA Rel. 3.1 (Opt) => IBM 4381, MVS/ESA Rel. 3.1 (Opt)

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 12 June 1991.

Compiler Name and Version: IBM Ada/370, Version 1.2.0 (with optimization)

Host Computer System: IBM 4381, MVS/ESA Rel. 3.1 (Optimized)

Target Computer System: IBM 4381, MVS/ESA Rel. 3.1 (Optimized)

Customer Agreement Number: 91-04-24-TEL

See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 910612W1.11167 is awarded to TeleSoft. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

for Ada Validation Organization
Director, Computer and Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

Accesion For	
NTIS CRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

DECLARATION OF CONFORMANCE

Customer:	TeleSoft 5959 Cornerstone Court West San Diego CA 92121
Certificate Awardee:	**International Business Machines Corporation**
Ada Validation Facility:	AVF, ASD/SCEL Wright-Patterson AFB, Ohio 45433-6503
ACVC Version:	1.11
Ada Implementation:	
Compiler Name and Version:	**IBM Ada/370, Version 1.2.0**
Host Computer System:	**IBM 4381** **(under MVS/ESA Release 3.1** **with Optimized Compiler)**
Target Computer System:	**Same as Host**

Declaration

We, the undersigned, declare that we have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed above.

Customer — Date: 4/25/91

TELESOFT
Raymond A. Parra, Director
Contracts/Legal

Certificate Awardee — Date: April 24'91

INTERNATIONAL BUSINESS MACHINES CORPORATION
Yim Chan, Ada Development Manager

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1), and possibly removing some inapplicable tests (see section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process, or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 3 May 1991.

E28005C	B28006C	C34006D	C35508I	C35508J	C35508M
C35508N	C35702A	C35702B	B41308B	C43004A	C45114A
C45346A	C45612A	C45612B	C45612C	C45651A	C46022A
B49008A	B49008B	A74006A	C74308A	B83022B	B83022H
B83025B	B83025D	C83026A	B83026B	C83041A	B85001L
C86001F	C94021A	C97116A	C98003B	BA2011A	CB7001A
CB7001B	CB7004A	CC1223A	BC1226A	CC1226B	BC3009B
BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E	CD2A23E
CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C	BD3006A
BD4008A	CD4022A	CD4022D	CD4024B	CD4024C	CD4024D
CD4031A	CD4051D	CD5111A	CD7004C	ED7005D	CD7005E
AD7006A	CD7006E	AD7201A	AD7201E	CD7204B	AD7206A
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3116A	CE3118A	CE3411B	CE3412B	CE3607B	CE3607C
CE3607D	CE3812A	CE3814A	CE3902B		

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 20 tests check for the predefined type LONG_INTEGER; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT; for this implementation, there is no such type.

C45423A, C45523A, and C45622A check that the proper exception is raised if MACHINE_OVERFLOWS is TRUE and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is FALSE.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

CA2009C and CA2009F check whether a generic unit can be instantiated before its body (and any of its subunits) is compiled; this implementation creates a dependence on generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3.)

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults; these instantiations are rejected by this compiler.

CE2401H uses instantiations of DIRECT_IO with unconstrained record types with discriminants with defaults; this implementation raises USE_ERROR on the attempt to create a file with such a type.

The tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	----------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The following 16 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE2107B..E CE2107G..H CE2107L CE2110B CE2110D

IMPLEMENTATION DEPENDENCIES

CE2111D CE2111H CE3111B CE3111D..E CE3114B
CE3115A

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST; for this implementation, the value of COUNT'LAST is greater than 150000, making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 28 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

BA1001A1	BA2001C	BA2001E2	BA3006A6M	BA3006B3
BA3007B7	BA3008A4	BA3008B5	BA3013A6	BA3013A7M

C52008B was graded passed by Test Modification as directed by the AVO. This test uses a record type with discriminants with defaults; this test also has array components whose length depends on the values of some discriminants of type INTEGER. The test was modified to constrain the subtype of the discriminants. Line 16 was modified to declare a constrained subtype of INTEGER, and discriminant declarations in lines 17 and 25 were modified to use that subtype; the lines are given below:

```
16   SUBTYPE SUBINT IS INTEGER RANGE -128 .. 127;
17   TYPE REC1(D1,D2 : SUBINT) IS

25   TYPE REC2(D1,D2,D3,D4 : SUBINT := 0) IS
```

CA2009C and CA2009F were graded inapplicable by Evaluation Modification as directed by the AVO. These tests contain instantiations of a generic unit prior to the compilation of that unit's body; as allowed by AI-00408 and AI-00506, the compilation of the generic unit bodies makes the compilation unit that contains the instantiations obsolete.

BC3204C and BC3205D were graded passed by Processing Modification as directed by the AVO. These tests check that instantiations of generic units with unconstrained types as generic actual parameters are illegal if the generic bodies contain uses of the types that require a constraint. However, the generic bodies are compiled after the units that contain the instantiations, and this implementation creates a dependence of the instantiating units on the generic units as allowed by AI-00408 and AI-00506 such that the compilation of the generic bodies makes the instantiating units obsolete--no errors are detected. The processing of these tests was modified by re-compiling the obsolete units; all intended

errors were then detected by the compiler.

CD1009A, CD1009I, CD1C03A, CD2A21C, CD2A24A, CD2A31A, CD2A31B, CD2A31C were graded passed by Evaluation Modification as directed by the AVO. These tests use instantiations of the support procedure LENGTH_CHECK, which uses Unchecked_Conversion according to the interpretation given in AI-00590. The AVO ruled that this interpretation is not binding under ACVC 1.11; the tests are ruled to be passed if they produce Failed messages only from the instances of LENGTH_CHECK--i.e, the allowed Report.Failed messages have the general form:

" * CHECK ON REPRESENTATION FOR <TYPE_ID> FAILED."

CE2203A and CE2403A were graded passed by Test Modification as directed by the AVO. These tests check that USE_ERROR is raised if the capacity of the external file is exceeded; but they require that the capacity can be limited to 4096 characters or less. This implementation can restrict file capacity in units of disk tracks, whose size is approximately 48K bytes. The two tests were revised at lines 45 & 54 (2 lines in each test) to use the form parameter "BLKSIZE 512 PRIMARY 1 SECONDARY 0", which restricts capacity to one track. The upper bound of the loop range at line 73 of each test was changed from 9 to 100, to attempt to write 50K bytes.

EE3301B, EE3405B, and EE3410F were graded passed by Evaluation Modification as directed by the AVO. These tests check certain I/O operations on the current default output file, including standard output. This implementation outputs the ASCII form-feed character which has no effect on the standard IBM output devices; in general, there is no common form-feed mechanism for the devices. Thus, the printed output from this test did not contain the expected page breaks. The AVO ruled that these tests should be considered passed if none of the tests' internal checks was failed (i.e., if the tests report "TENTATIVELY PASSED").

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical and sales information about this Ada implementation, contact:

IBM Canada, Ltd
844 Don Mills Road
North York, Ontario
Canada M3C IB7
ATTN: Antony Niro
31/257/844/TOR

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support

of a file system — if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3777
b) Total Number of Withdrawn Tests	94
c) Processed Inapplicable Tests	98
d) Non-Processed I/O Tests	0
e) Non-Processed Floating-Point Precision Tests	201
f) Total Number of Inapplicable Tests	299
g) Total Number of Tests for ACVC 1.11	4170

3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

ADA dsname {options}

options	description
dsname	Dsname specifies the file to be compiled.
ERROR(LIST)	Creates a listing file only when errors are encountered. The file contains compile-time error messages interspersed with the source code.
COMPILE | MAIN | BIND	Compile is the default option causing a compile only. BIND will be used in those instances for subunits needing to be compiled prior to the main program. MAIN is specified for mains and will allow execution to take place.
OPT(AUTO)	This option is used where optimized

results are desired.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	200 -- Value of V
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_646
$DEFAULT_MEM_SIZE	16777215
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	IBM370
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	ENT_ADDRESS
$ENTRY_ADDRESS1	ENT_ADDRESS1
$ENTRY_ADDRESS2	ENT_ADDRESS2
$FIELD_LAST	1000
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	CANNOT_RESTRICT_FILE_CAPACITY
$GREATER_THAN_DURATION	86401.0
$GREATER_THAN_DURATION_BASE_LAST	131073.0
$GREATER_THAN_FLOAT_BASE_LAST	7.237006E+75
$GREATER_THAN_FLOAT_SAFE_LARGE	7.23004E+75

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        7.237E+75

$HIGH_PRIORITY          255

$ILLEGAL_EXTERNAL_FILE_NAME1
                        BADCHAR*%

$ILLEGAL_EXTERNAL_FILE_NAME2
                        BAD-CHAR!@~

$INAPPROPRIATE_LINE_LENGTH
                        1029

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.TST");

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006F1.TST");

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483647

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     C

$LESS_THAN_DURATION     -86401.0

$LESS_THAN_DURATION_BASE_FIRST
                        131073.0

$LINE_TERMINATOR        ' '

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        NULL;

$MACHINE_CODE_TYPE      NO_SUCH_TYPE

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                -2147483648

$NAME                   NO_SUCH_TYPE_AVAILABLE
```

MACRO PARAMETERS

$NAME_LIST	mc68000,anuyk44,ibm370
$NAME_SPECIFICATION1	'TEST3.X2102A'
$NAME_SPECIFICATION2	'TEST3.X2102B'
$NAME_SPECIFICATION3	'TEST3.X3119A'
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	16777215
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	IBM370
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	NEW INTEGER;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	1024
$TICK	0.000001
$VARIABLE_ADDRESS	VAR_ADDRESS
$VARIABLE_ADDRESS1	VAR_ADDRESS1
$VARIABLE_ADDRESS2	VAR_ADDRESS2
$YOUR_PRAGMA	PRIORITY

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

Chapter 2. Compiling Ada Programs

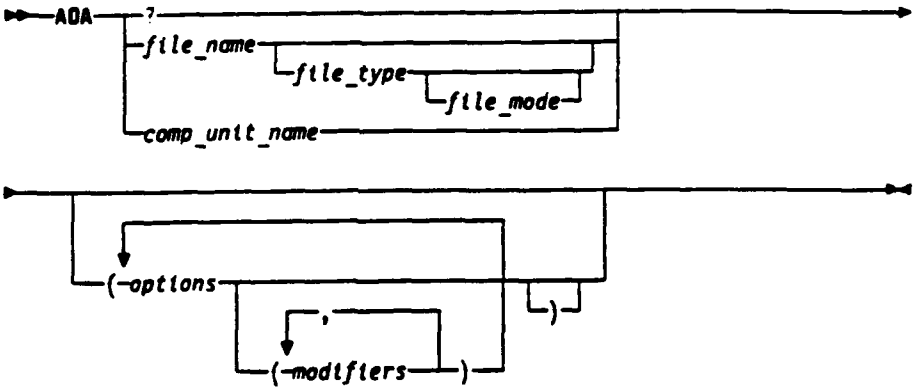
This chapter describes how to use the IBM Ada/370 compiler. You should have available a copy of *IBM Ada/370 Language Reference* (SC09-1297) or ANSI/MIL-STD-1815A, the Ada LRM.

If you need help getting started with IBM Ada/370, see Chapter 9. IBM Ada/370 Tutorial.

Compiling a Source Program

The ADA command compiles a source program. The following sections show you how to use this command under VM/CMS, MVS TSO, and as an MVS batch job.

Using the ADA Command under VM/CMS



The "?" option displays syntax information, including a list of the ADA options, on the screen.

Most situations require that you pass the file name of the source file. The compilation unit name is required when you use the Bind option, or when you use the Run and NOCOmpile options. The *file_type* and *file_mode* default to ADA and *, respectively.

When you specify a compiler option in ADA, you can use the minimum unique abbreviation. For example, you can specify CReate as CR.

Many compiler options are matched by an opposite. For example, the opposite of the MAP option is the NOMAP option. For such cases, one of the options is designated as the default. For an option that takes a numerical value, a particular value may be assigned as the default. The compiler uses default settings unless you override them by specifying the nondefault options to ADA.

Precede the list of options by a blank space and a left parenthesis, and separate them from each other by blank spaces. A closing parenthesis is optional.

Some of the options have modifiers, which you must enclose in parentheses. Where you can enter multiple modifiers to an option, such as with the Xref option, separate the modifiers with a comma.

Here are some examples:

1. ADA EXAMPLE

   Example 1 compiles EXAMPLE ADA * using the default options.

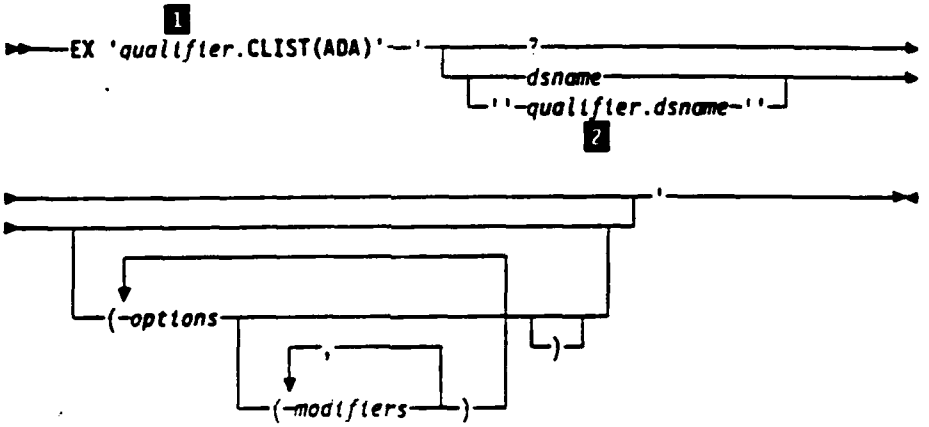
2. ADA EXAMPLE (LIBRARY (DEMO LIBRARY) DEBUG

   Example 2 compiles EXAMPLE ADA * with the LIBrary and Debug options. LIBrary causes the compiler to use a file containing an alternative library. The library has the name DEMO LIBRARY *.

3. ADA EXAMPLE (XREF (BYUNIT,FULL)

   Example 3 compiles EXAMPLE ADA * with the Xref option to produce a cross-reference listing. The listing is ordered by compilation unit and includes cross-references to all visible units.

Using the ADA Command under MVS TSO



Under MVS, the *qualifier* at 1 is the high-level qualifier assigned to IBM Ada/370 by your system administrator. In examples for this book, we use the high-level qualifier ADA. The *qualifier* at 2 is the high-level qualifier for the data set you are specifying. By default, the high-level qualifier is set by the PROFILE PREFIX command. This, in turn, defaults to your TSO logon identifier.

The "?" option displays syntax information, including a list of the ADA options, on the screen.

Most situations require that you pass the file name of the source file. The compilation unit name is required when you use the Bind option.

When you specify a compiler option in ADA, you can use the minimum unique abbreviation. For example, you can specify CReate as CR.

Many compiler options are matched by an opposite. For example, the opposite of the MAP option is the NOMAP option. For such cases, one of the options is designated as the default. For an option that takes a numerical value, a particular value may be assigned as the default. The compiler uses default settings unless you override them by specifying the nondefault options to ADA.

Precede the list of options by a blank space and a left parenthesis, and separate them from each other by blank spaces. A closing parenthesis is optional.

Some of the options have modifiers, which you must enclose in parentheses. Where you can enter multiple modifiers to an option, such as with the Xref option, separate the modifiers with a comma.

Here are some examples:

1. EX 'ADA.CLIST(ADA)' 'EXAMPLE'

   Example 1 compiles EXAMPLE using the default options.

2. EX 'ADA.CLIST(ADA)' 'EXAMPLE (LIBRARY (''DEMO.LIBRARY'') DEBUG'

   .Example 2 compiles EXAMPLE with the LIBrary and Debug options. LIBrary causes the compiler to use a data set containing an alternative library. The library has the name DEMO.LIBRARY. The compiler also saves information needed by the IBM Ada/370 debugger.

3. EX 'ADA.CLIST(ADA)' '''USER1.EXAMPLE'' (XREF (B,UNIT,FULL)'

   Example 3 compiles EXAMPLE with the high-level qualifier USER1. It includes the Xref option to produce a cross-reference listing. The listing is organized by compilation unit and includes cross-references to all visible units.

Compiling a Program with Job Control Language (JCL)

This section describes how to invoke the compiler as a batch job under MVS using Job Control Language (JCL). For information on how to invoke the binder using JCL, see "Invoking the Binder with Job Control Language (JCL)" on page 3-2.

The ADAC cataloged procedure invokes the compiler on a source file.

```
//MYPROG    JOB ,'  ',MSGCLASS=0,MSGLEVEL=(1,1),NOTIFY=USER1,
//       CLASS=A
//*
//*  PURPOSE:  TO RUN THE ADA COMPILER
//*
//COMPILE  EXEC PROC=ADAC,ADASRC='USER1.ADA.SOURCE(HELLO)',
//         USER=USER1, CMPPRM='CHECK'
```

Figure 2-1. Using the ADAC Cataloged Procedure to Invoke the Binder

The preceding example job, MYPROG, compiles member HELLO in the source PDS, USER1.ADA.SOURCE. The user's name, USER1, is identified with the USER variable. This variable is used as a high-level qualifier to construct data-set names for the compiler, such as USER1.ADA.LIBRARY, which is the default library. Your job card will probably be different, because it depends on your site's conventions.

After you execute this job, the Ada program contained in member HELLO is compiled into the working sublibrary of USER1.ADA.LIBRARY.

A sample of ADAC cataloged procedure appears in Figure 2-2 on page 2-4. The exact location of ADAC may depend on your site's conventions.

```
//ADAC     PROC CMPPRM=' ',MEMSIZE=8196K,
//          STPLIB='ADA110.LOADLIB',MAXTIME=60,
//          VIO=VIO,SYSDA=SYSALLDA,SYSOUT='*'
//*
//* ERASE ADA.INFO DATASET
//*
//          EXEC PGM=IEFBR14
//ADAINFO   DD  DSN=&USER..ADA.INFO,DISP=(MOD,DELETE),
//          SPACE=(1,1),UNIT=&SYSDA
//*****************************************************************
//*                      INVOKE THE COMPILER
//*****************************************************************
//STEP1     EXEC PGM=EVGCOMP,PARM='&CMPPRM',REGION=&MEMSIZE,
//          TIME=&MAXTIME,DYNAMNBR=65
//STEPLIB   DD  DSN=&STPLIB,DISP=SHR
//CONOUT    DD  SYSOUT=&SYSOUT,DCB=(LRECL=120,BLKSIZE=120)
//ADAIN     DD  DSN=&ADASRC,DISP=SHR,FREE=END,DCB=BUFNO=4
//ADALIB    DD  DSN=&USER..ADA.LIBRARY,DISP=SHR
//ADAINFO   DD  DSN=&USER..ADA.INFO,DISP=(NEW,PASS,CATLG),
//          DCB=(RECFM=VB,LRECL=512,BLKSIZE=3120,DSORG=PS),
//          SPACE=(80,(10,50)),UNIT=&SYSDA
//ADALIST   DD  DSN=&USER..LISTING,DISP=(MOD,CATLG,CATLG),
//          DCB=(RECFM=VBA,LRECL=259,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//          SPACE=(132,(500,2000,20)),UNIT=&SYSDA
//ADAUT1    DD  SPACE=(132,(500,2000)),
//          DCB=(RECFM=FB,LRECL=136,BLKSIZE=3400,DSORG=DA,BUFNO=2),
//          UNIT=&SYSDA
//ADAUT2    DD  SPACE=(132,(500,2000)),
//          DCB=(RECFM=VB,LRECL=136,BLKSIZE=3120,DSORG=PS,BUFNO=2),
//          UNIT=&VIO
//ADAUT3    DD  SPACE=(132,(500,2000)),
//          DCB=(RECFM=FB,LRECL=1028,BLKSIZE=2056,DSORG=DA,BUFNO=2),
//          UNIT=&SYSDA
//ADAUT4    DD  SPACE=(132,(500,2000)),
//          DCB=(RECFM=FB,LRECL=132,BLKSIZE=2640,DSORG=DA,BUFNO=2),
//          UNIT=&SYSDA
```

Figure 2-2. ADAC Cataloged Procedure

Symbolic Variables for ADAC Cataloged Procedure

The ADAC cataloged procedure includes several symbolic JCL substitution variables you can modify to specify the various options available.

Symbolic Variable	Description
ADASRC	Specifies the data-set name (DSN) for the Ada source file. This name must be set in order to successfully compile an Ada program.
CMPPRM	Specifies options to the compiler in the PARM field. This variable, found in Step 1 of the ADAC cataloged procedure, specifies options to the compiler in the PARM field. These options are the same ones used when you invoke the compiler with the ADA command.

The compiler options have the following syntax:

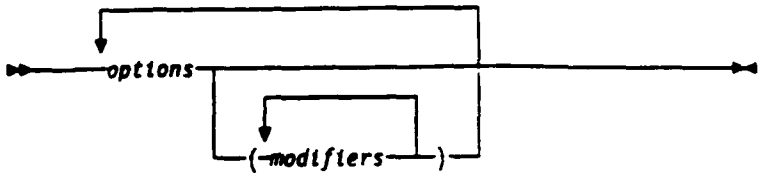


Table 2-2 on page 2-8 lists the valid options, their modifiers, and defaults.

MAXTIME — Sets a maximum amount of time for the compiler job step to run (via the TIME parameter on a JCL EXEC card). The default value is 60 minutes, but this value can be changed when the cataloged procedure is installed.

MEMSIZE — Specifies the amount of memory in which to run the compiler job step (via the REGION option on a JCL EXEC card). The greater the memory, the better the performance of the compiler. The default value is 8196Kb, but this value may be changed, depending on your site's conventions.

STPLIB — Indicates the data-set name of the partitioned data set that contains the compiler module, EVGCOMP. This default can be changed when the cataloged procedure is installed.

SYSDA — Specifies the UNIT for permanent data set allocations. The default is SYSALLDA, but this value can be changed depending on your site's conventions.

SYSOUT — Identifies the output class for the compiler output. The default is "*", but this can be changed, depending on your site's conventions.

USER — Indicates the high-level qualifier that is required to build data-set names used by the compiler. You must specify this variable. It is common to set this variable to your TSO logon identifier.

VIO — Specifies the UNIT for temporary data set allocations. The default is VIO, but the default can be changed depending on your site's conventions.

The Compiler Options

The ADA command invokes the IBM Ada/370 compiler. Table 2-1 provides a brief summary of the compiler options. The square brackets enclose optional modifiers. You do not actually enter the brackets as part of the command syntax. For the specific syntax of each option, see the option descriptions on the pages specified in the table.

Table 2-1 (Page 1 of 2). Compiler Options

Option	Default	Function	Page
Asm	NOAsm	Assembly listing.	2-9
[NOGen		Suppress listing of expanded generics.	
| NOSys]		Suppress listing of system-supplied generics.	
Bind	COmpile	Bind previously-compiled main unit.	2-10
CHeck	COmpile	Compile with syntactic and semantic checking only.	2-10
[Nosemantic]		Compile with syntactic checking only.	
COmpile	COmpile	Compile code for a library unit.	2-11
CReate [*number_of_units*]	NOCReate	Initialize working sublibrary for the compiler. *number_of_units* is number of compilation units in sublibrary.	2-11
Debug	NODebug	Output information for debugging.	2-13
Error	NOError	Specify action to be taken when errors occur. Must include at least one modifier.	2-13
[Count = *number*]		Abort compilation after *number* errors.	
[List]		Generate interspersed listing of errors and source code.	
Generate	NOGenerate (VM/CMS)	Generate a load image.	2-14
NOGenerate	Generate (MVS)		
INlist [*max_number*]	COmpile	Compile multiple source files with one invocation of the compiler. *max_number* is the maximum number of compilation failures during input list processing.	2-15
LIBrary *library_name*		Specify Ada library name.	2-15
LISt	NOLISt	Generate interspersed listing of errors and source code.	2-16
MAIn [*comp_unit_name*]	COmpile	Compile and bind code for a main unit.	2-16
MAP	NOMAP	Produce linkage map during binding. Use with MAIn, Bind, or Run options.	2-17
NOCOmpile	COmpile	Suppress the compiler.	2-17
Run	NORun	Execute main program.	2-18

Table 2-1 (Page 2 of 2). Compiler Options

Option	Default	Function	Page
Suppress	NOSuppress	Suppress selected run-time checks or line information tables in generated object code. Must include at least one modifier.	2-18
[Lineinfo]		Suppress generation of line information tables.	
[Checks		Suppress all run-time checks.	
[Elab]		Suppresses only elaboration checks.	
Trace	NOTrace	Display diagnostic messages from the compiler.	2-19
Xref	NOXref	Produce a cross-reference listing.	2-19
[Byunit]		Order the listing by compilation unit.	
[Full]		Cross-reference all visible units.	

Compiler Options for Use with ADAC Cataloged Procedure

This section defines the standard options for the CMPPRM symbolic variable when you invoke the compiler as a batch job under MVS. For more information on CMPPRM, see "Symbolic Variables for ADAC Cataloged Procedure" on page 2-4. Descriptions of the options appear on the pages shown.

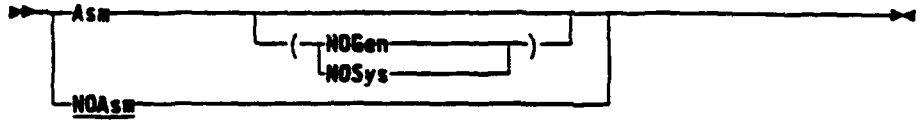
Table 2-2. Compiler Options for JCL

Option	Default	Function	Page
ASM	NOASM	Assembly listing.	2-9
[NOGEN		Suppress listing of expanded generics.	
| NOSYS]		Suppress listing of system-supplied generics.	
CHECK	COMPILE	Compile with syntactic and semantic checking only.	2-10
[NOSEMANTIC]		Compile with syntactic checking only.	
COMPILE	NOCOMPILE	Compile code for a library unit.	2-11
CREATE [*number_of_units*]	NOCREATE	Initialize working sublibrary for the compiler. *number_of_units* is number of compilation units in sublibrary.	2-11
DDNAMES *old_name = new_name*		Specify the Data Description (DD) names that identify the data sets used by the compiler and binder. It must include at least one modifier.	2-12
DEBUG	NODEBUG	Output information for debugging.	2-13
ERROR	NOERROR	Specify action to be taken when errors occur. Must include at least one modifier.	2-13
[COUNT = *number*]		Abort compilation after *number* errors.	
[LIST]		Generate interspersed listing of errors and source code.	
LIST	NOLIST	Generate interspersed listing of errors and source code.	2-16
SUPPRESS	NOSUPPRESS	Suppress selected run-time checks or line information tables in generated object code. Must include at least one modifier.	2-18
[LINEINFO]		Suppress generation of line information tables.	
[CHECKS		Suppress all run-time checks.	
| ELAB]		Suppresses only elaboration checks.	
TRACE	NOTRACE	Display diagnostic messages from the compiler.	2-19
XREF	NOXREF	Produce a cross-reference listing.	2-19
[BYUNIT]		Order the listing by compilation unit.	
[FULL]		Cross-reference all visible units.	

Detailed Descriptions of Compiler Options

The following detailed descriptions of the compiler options include syntax diagrams. In these diagrams, uppercase characters indicate the minimum abbreviation of options and their modifiers. Options and modifiers that are underscored are the defaults.

Asm Option



The Asm option produces pseudo-assembly language for the object code interspersed with the Ada source for each compilation unit. It causes the creation one listing for each source file. Asm also provides information on the relative offset and size allocation of each data item or constant. This listing is called the data map. For more information on the listing produced by Asm, see "Source and Assembly Listings" on page 7-1.

If you use the Asm option at the same time you invoke the IBM Ada/370 binder, the compiler also produces a binder listing.

The NOGen modifier suppresses the listing of code generated for expanded generics. Otherwise, listings include the code generated for all expanded generics.

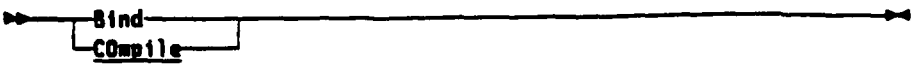
The NOSys modifier suppresses the listing of code generated for system-supplied generics.

Use NOGen or NOSys to reduce the size of listings.

Under VM/CMS
The name of the listing file takes the form *source* LISTING A, where *source* is the file name of the source file.

Under MVS
The name of the listing data set takes the form *qualifier*.LISTING(*source*) where *qualifier* is the TSO logon identifier and *source* can be either the name of the member of a partitioned data set (PDS) used as source or the second qualifier in the name of a sequential data set.

Bind Option



The Bind option binds a main program that has been previously compiled as a library unit. As output, it produces an object file. When you use this option, enter the compilation unit name in place of the source file name. You can only use the Bind option for compilation units that reside in the working sublibrary of the Ada program library.

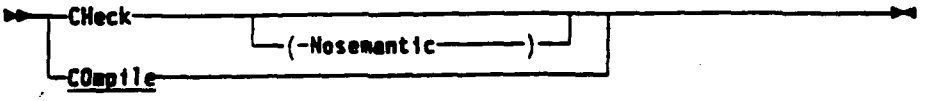
To invoke the binder when you compile the source (rather than in a separate call to ADA), use the MAIn option.

You cannot use the Bind option in combination with MAIn, CHeck, INlist, COmpile, or NOCOmpile. If you enter more than one of these options, IBM Ada/370 only accepts the last one in the command line. If you do not use any of these options, the default is COmpile.

Under VM/CMS
The object file created by compiling with the Bind option has the file type TEXT.

Under MVS
The object file is a PDS created by compiling with the Bind option. It takes the form *qualifier*.OBJ(*comp_unit*), where *qualifier* is your TSO logon identifier and *comp_unit* is the compilation unit name.

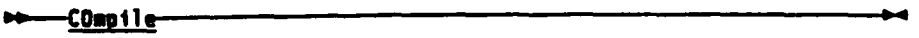
Check Option



The CHeck option causes the compiler to perform only syntactic and semantic error checking. Because no object code is produced, you can save compilation time and disk space during error checking. If you include the Nosemantic modifier, the compiler only performs syntactic error checking.

You cannot use the CHeck option in combination with MAIn, Bind, INlist, COmpile, or NOCOmpile. If you enter more than one of these options, IBM Ada/370 only accepts the last one in the command line. If you do not use any of these options, the default is COmpile.

Compile Option

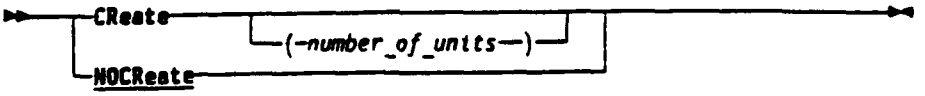


The COmpile option causes all compilation units in the source file to be library units, rather than main units.

You can also make a library unit into a main unit using the Bind option.

You cannot use the COmpile option in combination with MAIn, Bind, CHeck, INlist, or NOCOmpile. If you enter more than one of these options, IBM Ada/370 only accepts the last one in the command line. If you do not use any of these options, the default is COmpile.

Create Option



CReate initializes the working sublibrary for the compiler. The compiler creates a new sublibrary, deleting the previous copy, if one exists.

The *number_of_units* variable specifies the number of compilation units the sublibrary can contain. The default is 200. The largest number of units a sublibrary can contain is 4671.

This number indicates an approximate size for the sublibrary. The number of units that actually fit into a sublibrary depends upon their size and complexity. For further information on sublibraries, see Chapter 5, Working with the Ada Library System.

When you use CReate in conjunction with the LIBrary option, it initializes the working sublibrary in the library specified by LIBrary.

You cannot use CReate in combination with the Bind option.

NOCReate is the default. With NOCReate, the compiler does not initialize the working sublibrary.

DDNAMES Option (MVS JCL Only)

DDNAMES specifies the Data Description (DD) names that identify the data sets used by the compiler and binder. DDNAMES always requires a value.

For use with the compiler, *old_name* has one of the following values:

ADAIN
ADAINFO
ADALIB
ADALIST
ADAUT1
ADAUT2
ADAUT3
ADAUT4

For use with the binder, *old_name* has one of the following values:

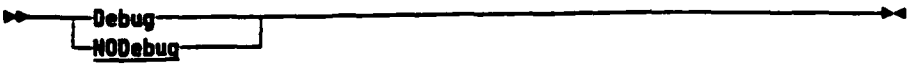
ADAINFO
ADALIB
ADALIST
ADAUT1
ADAUT2
ADAUT3
ADAUT4

Usually, you do not need to change the DD names associated with the compiler.

This example of a JCL code fragment uses DDNAMES to identify a new DD name specifying the source file and the ada library:

```
//MYSTEP   EXEC PGM=EVGCOMP,PARM='DDNAMES(ADAIN=MYSOURCE,ADALIB=MYLIB)'
//MYSOURCE DD   DSN=qualifier.ADA.SOURCE(PROGRAM1)
//MYLIB    DD   DSN=qualifier.ADA.LIBRARY
```

Debug Option



The Debug option causes information used by the IBM Ada/370 debugger to be placed in the working sublibrary. When used with the MAIn or Bind options, Debug produces a debugging map, which is required by the debugger.

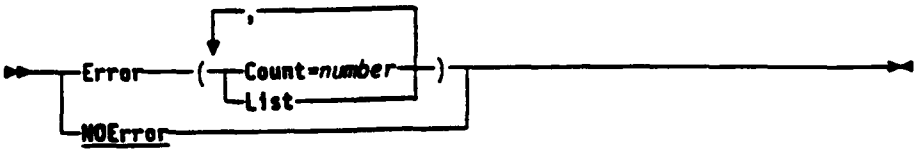
For more information about debugging, see Chapter 8. The IBM Ada/370 Debugger.

With NODebug, the compiler does not place debugging information into the working sublibrary.

Under VM/CMS
The debug map file takes the name *comp_unit* DEBUGMAP A where *comp_unit* is derived from the name of the compilation unit.

Under MVS
The debug map file takes the name *qualifier*.DEBUGMAP(*comp_unit*), where *qualifier* is the high-level qualifier and *comp_unit* is derived from the name of the compilation unit.

Error Option



Error controls the way the compiler behaves when it finds errors in the source file. You must choose at least one of the modifiers.

The Count modifier specifies the *number* of errors that cause the compiler to stop processing. The compiler includes syntax, semantic, and warning errors in the count. For example, COUNT=5 causes the compiler to stop processing after it finds five errors. If you omit the Count modifier, the compiler stops processing when it finds 32767 errors, the default error limit.

The List modifier creates a file containing compile-time error messages interspersed with the source code. If there are no errors, the compiler does not generate the listing. To generate this listing regardless, use the LISt option. See "List Option" on page 2-16.

With NOError, the compiler does not modify its behavior when it finds errors during processing.

Under VM/CMS
The listing file created by the List modifier is called *source* LISTING A, where *source* is the file name of the source file.

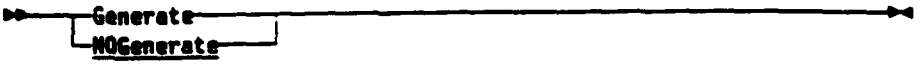
Under MVS

The listing file created by the LISt modifier is called *qualifier*.LISTING(*source*), where *qualifier* is the TSO logon identifier and *source* is the file name of the source data set.

Generate Option

The Generate option generates a load image.

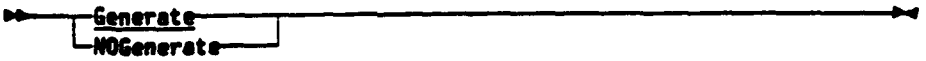
Under VM/CMS



Under VM/CMS, when you compile with the Generate option, the ADA command also invokes the binder, producing an object file. The compiler then uses the object file to produce the load module. This option assumes that the source file contains a main program. The load module created by the Generate option has the name *comp_unit* MODULE A, where *comp_unit* is derived from the compilation unit name.

With NOGenerate, the compiler does not generate an executable load module.

Under MVS



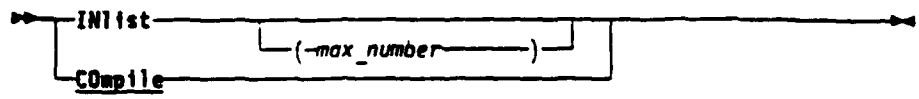
The NOGenerate option suppresses the invocation of the linkage editor after binding a main program. Use this option when you want to invoke the linkage editor with options that differ from the default. For example, you would use NOGenerate when you want to link the main program with non-Ada object code.

The load module created by the Generate option has the name *qualifier*.LOAD(*comp_unit*), where *comp_unit* is derived from the name of the compilation unit.

The NOGenerate option has no effect when used in combination with NOCOmpile, or when you compile the program as a library unit instead of a main unit.

You cannot use NOGenerate in combination with the Run option.

Inlist Option



The INlist option compiles multiple source files with a single invocation of the compiler. When you use this option, enter the name of the file containing the input list in place of the source file name. For more information on the use of input lists, see "Compiling Multiple Source Files" on page 2-20.

If a source file fails to compile, the compiler continues to process the remaining files. You can specify that the compiler stops processing after a certain number of source files fail to compile. To do so, use the *max_number* variable.

You cannot use INlist in combination with the Bind, CHeck, MAIn, COmpile or NOCOmpile options. If you enter more than one of these options, IBM Ada/370 only accepts the last one in the command line. If you do not use any of these options, the default is COmpile.

Library Option

The LIBrary option specifies the name of the Ada library file to be used by the compiler. The *library_name* modifier is the name of a library file that contains the names of one or more sublibraries.

When you do not specify the LIBrary option, the compiler uses the default library file. Under VM/CMS, it has the name ADA LIBRARY *. Under MVS, it has the name *qualifier* ADA.LIBRARY, where *qualifier* is the TSO logon identifier. For information concerning libraries and sublibraries, see Chapter 5, Working with the Ada Library System.

Under VM/CMS

You can provide the *library_name* variable in either of two formats. The preferred is *file_name file_type file_mode*. The other format is *file_mode:file_name.file_type*. In both formats, if you specify only the file name, *file_type* defaults to LIBRARY and *file mode* defaults to "*". If you do not specify a library file, ADA searches for ADA LIBRARY *.

For example, to specify library PROJ1 LIBRARY A, when you compile the file MYPROG, enter:

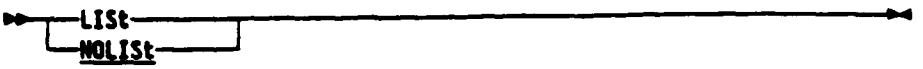
```
ADA MYPROG (LIB(PROJ1))
```

It is recommended that you do not select an alternative file type for the Ada library file. Retaining the default file type maintains consistent file naming conventions for all users.

Under MVS
A library can be either a sequential data set or a member of a PDS. The *library_name* variable can be any valid data-set name format. For example, to specify library PROJ1.LIBRARY on USER1 when you compile the data set MYPROG.SOURCE enter:

```
EX 'ADA.CLIST(ADA)' '''USER1.MYPROG.SOURCE'' (LIB(''USER1.PROJ1.LIBRARY'')'
```

List Option

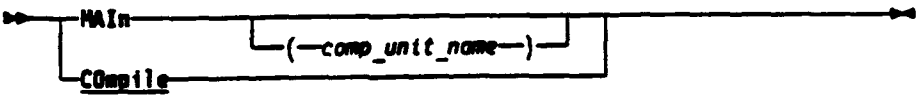


The LISt option creates a file containing a listing for each source file. The listing contains compile-time messages interspersed with the source code. If there are multiple compilation units in a source file, LISt places the listings for all units into a single file.

Under VMICMS
The compilation listing file takes the form *source* LISTING A, where *source* is the file name of the source file.

Under MVS
The compilation listing file takes the form *qualifier*.LISTING(*source*), where *qualifier* is the TSO logon identifier and *source* can be either the name of the member of a PDS used as the source or the second qualifier in the name of a sequential data set.

Main Option



The MAIn option causes the compiler to produce code for the source file as an Ada main program. The MAIn option compiles a program and performs the binding operation without the need to specify any other option.

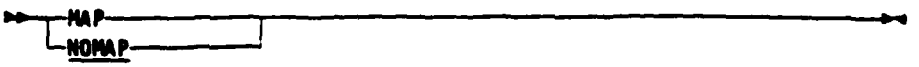
If the Ada source file contains one or more library compilation units in addition to the main compilation unit, enter the name of the main program in the *comp_unit_name* variable.

You cannot use the MAIn option in combination with Bind, CHeck, INlist, COmpile, or NOCOmpile. If you enter more than one of these options, IBM Ada/370 only accepts the last one in the command line. If you do not use any of these options, the default is COmpile.

Under VMICMS
The object file created by compiling with the MAIn option takes the form *comp_unit* TEXT A, where *comp_unit* is the compilation unit name.

Under MVS
The object file created by compiling with the MAIn option takes the form *qualifier*.OBJ(*comp_unit*), where *qualifier* is the TSO logon identifier and *comp_unit* is the compilation unit name.

Map Option



The MAP option causes the compiler to produce a linkage map when the IBM Ada/370 binder processes a main program.

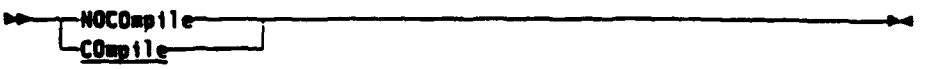
Use MAP in combination with either MAIn or Bind, both of which invoke the binder. You can also use MAP in combination with Run as long as you do not use the NOCOmpile option.

With NOMAP, the compiler does not create a linkage map when the IBM Ada/370 binder processes a main program.

Under VM/CMS
The map file is called *object* ADAMAP A, where *object* is the file name of the object file.

Under MVS
The map file is called *qualifier* ADAMAP(*comp_unit*), where *qualifier* is the TSO logon identifier and *comp_unit* is the name of the main compilation unit.

Nocompile Option

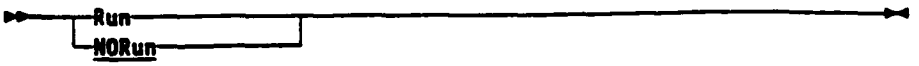


The NOCOmpile option causes the ADA command to suppress the compilation step. Thus, you can use NOCOmpile with Run to run an Ada program that has already been compiled. When you use NOCOmpile with Run, enter the compilation unit name in place of the source file name.

You can also use NOCOmpile with the CReate to create a new working sublibrary without having to compile the source code.

You cannot use the NOCOmpile option in combination with MAIn, Bind, CHeck, INlist, or COmpile. If you enter more than one of these options, IBM Ada/370 only accepts the last one in the command line. If you do not use any of these options, the default is COmpile.

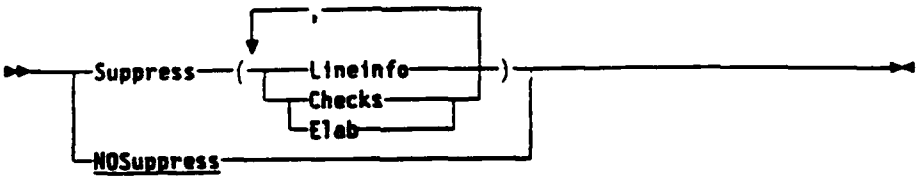
Run Option



The Run option loads and executes a main program. The compiler assumes that the program is a main unit. You can either compile and run a program, or run a precompiled program. To run a previously compiled and bound program, use Run in combination with the NOCOmpile option. When you use Run with NOCOmpile, you must specify the compilation unit name, rather than the Ada source file name.

With NORun, the compiler does not execute the program.

Suppress Option



The Suppress option suppresses selected run-time checks and line information in generated object code, resulting in smaller, faster modules. You must choose at least one of the modifiers. Use of either the Suppress option or pragma Suppress causes the compiler to suppress run-time checks. For more information on pragma Suppress, see the chapter on tuning in the *IBM Ada/370 Programmer's Guide*.

The Lineinfo modifier suppresses the generation of line information tables, thus saving the space required to produce them. These tables display the Ada source line number when an unhandled exception occurs. If you compile your code with this option and an unhandled exception occurs during run time, the error information does not include a line number.

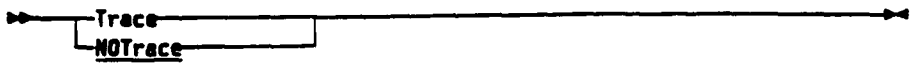
The Checks modifier suppresses all run-time checks.

The Elab modifier only suppresses elaboration checks made by other units on this unit. This differs from the way pragma Suppress works. The pragma suppresses elaboration checks made on other units from the unit in which it resides.

If you choose both the Checks and Elab modifiers, the Checks modifier takes precedence.

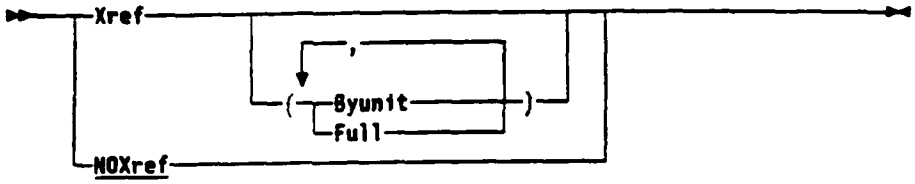
With NOSuppress, the compiler does not suppress selected run-time checks and line information in generated object code.

Trace Option



The Trace option displays diagnostic messages from the compiler. This option is intended for use in submitting problems to IBM. For more information, see the *IBM Ada/370 Diagnosis Guide*.

Xref Option



The Xref modifier produces a cross-reference listing for each compilation unit contained in the source file. It creates one listing file for each source file.

Byunit causes Xref to display symbols by compilation unit. By default, the Xref listing displays symbols in alphabetic order.

Full causes Xref to cross-reference each compilation unit with all unit specifications that are visible to it. A unit specification is visible if it is an import to the compilation unit. If the compilation unit is a body, its parent and its parent's imports are also visible. Full does not display cross references for the private parts of imported units. By default, Xref only cross-references the compilation units specified in the command. For more information, see "Cross-Referencer" on page 7-2.

VM/CMS Usage
The listing file takes the form *source* LISTING A, where *source* is the file name of the source file.

MVS Usage
The listing file takes the form *qualifier*.LISTING(*source*), where *qualifier* is the TSO logon identifier and *source* can be either the name of the member of a partitioned data set (PDS) used as source or the second qualifier in the name of a sequential data set.

Compiling Multiple Source Files

An input list is a file containing a list of the names of files to be compiled. Using input lists, you can compile multiple source files with a single invocation of the compiler. This reduces the time it takes to compile a group of source files because it eliminates some redundant activities within the compiler.

The names of source files appear in the input list, along with other information that controls the compilation process. The compiler processes items in the input list in sequential order. Besides the object code that is the usual result of compilation, the compiler produces a file that contains information on the results of the success or failure of each compilation.

If you use a compiler option that produces compilation listings (`Asm`, `LISt`, `Error`, or `Xref`, the compiler produces a separate listing for each compilation unit.

To compile multiple files with the ADA command, use the `INlist` option.

If the compiler detects errors during compilation of any source file in the list, it goes on to the next source file. There may be cases, especially with a large input list, where it is not advisable to continue through the entire input list when multiple source files abort. The `INlist` option has a variable that allows you to specify the maximum number of source file compilation failures to allow. The next failure causes the compiler to stop processing the input list.

Under VM/CMS
The command string

```
ADA MYLIST (IN(6)
```

compiles the source files in the input list MYLIST INLIST, setting the failure limit at six.

Under MVS
The command string

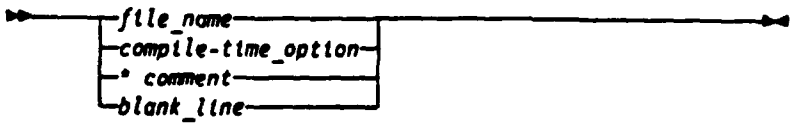
```
EX 'ADA.CLIST(ADA)' 'MYLIST.INLIST (IN(6)'
```

compiles the source files in the input list MYLIST.INLIST, setting the failure limit at six. The compiler uses the default high-level qualifier.

Constructing the Input List

An input list contains two types of information, names of source files to be compiled and compile-time options.

The syntax for each line of the input list is:



The rules for creating the input list are:

- Place only put one file name or compile-time option on a line.
- Do not place comments on the same line with other information.
- If the first nonblank character in a line is "*", that line is a comment line.

- The compiler ignores blank lines.
- File names do not have to start in the first column.

This VM/CMS file list follows the rules correctly.

```
* YES, THIS IS A COMMENT

AFILE ADA A
BFILE ADA A

   B1SUB ADA A
   B2SUB ADA A

CFILE ADA A
```

If an error occurs during input list processing, the compiler updates the working sublibrary with information about the units that have been compiled successfully. Also, the compiler places information about the results in an output file. For more information about the contents of this file, see "Getting Information on an Input List Compilation" on page 2-22.

VM/CMS Source File Names

Enter the names of source files into the input list. If you leave out the file type, the compiler assumes the file type is ADA.

Input List	Compiler Interpretation
MYFILE	MYFILE ADA *
MYFILE TEST	MYFILE TEST *
MYFILE TEST A	MYFILE TEST A

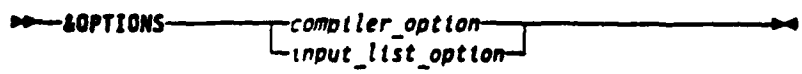
MVS Source Data Set Names

Enter the names of source data sets into the input list either fully or partially qualified. Enclose fully qualified names inside a pair of single quotation marks. If you leave out the high-level qualifier, the compiler assumes the current TSO logon identifier.

Input List	Compiler Interpretation
MYFILE	'*qualifier*.MYFILE'
MYFILE.TEST	'*qualifier*.MYFILE.TEST'
'OTHER.MYFILE.TEST'	'OTHER.MYFILE.TEST'

Embedding Compile-Time Options in the Input List

You can place compile-time options and input list options within the input list. Options set when you invoke the compiler apply to each source file until they are overridden by an option embedded in the input list. Options that appear in the input list apply to all following source files until they are overridden by other embedded options. A specific option can appear multiple times in an input list.



The compiler options you can embed in input lists are:

ASM
CHECK
COMPILE
DEBUG
ERROR
INLIST
LIST
SUPPRESS
XREF

There is one valid *input_list_option*, DEFault. The DEFault option causes the compiler to reset all options to their states as set by the ADA command. DEFault is only valid as part of the &OPTIONS command in an input list.

The following example shows an input list with embedded options, along with descriptions of how the options change. This example uses VM/CMS file naming conventions; MVS users should use MVS conventions.

Input List	How Options Change
AFILE ADA A &OPTIONS DEBUG	Command-line options
BFILE ADA A &OPTIONS NODEBUG	Command-line options plus Debug
CFILE ADA A &OPTIONS DEFAULT	Command-line options plus NODebug
EFILE ADA A &OPTIONS DEBUG	Command-line options only
FFILE ADA A	Command-line options plus Debug

Getting Information on an Input List Compilation

The compiler creates a file and places information about of the compilation into it. Each line in the input list also appears in this file. Following each line from the input list containing the name of a source file is a line that shows compilation status for the file. Source files that compile successfully show a return code of zero for each compilation unit in the file. Files that do not compile show the return code of the error that caused the failure. There are also descriptive messages where return codes do not provide enough information.

The return codes that can appear are:

Code	Explanation
0	Execution complete. No errors occurred.
4	Execution complete. Warnings were issued, but no errors occurred.
8	Source code errors, such as syntactic or semantic errors, were detected. Look for specific errors in the console listing.

Below is a brief example of an input list and the OUTPUT file that might result. This example uses VM/CMS file naming conventions; under MVS the output follows MVS conventions.

Sample Input List MYLIST INPUT A

```
&OPTIONS ASM
FILEONE ADA A
&OPTIONS DEF
FILETWO ADA A
&OPTIONS BLTZ
```

Sample OUTPUT File

```
INPUT LIST processing MYLIST INPUT A - yyyy-mmm-dd hh:mm:ss - options (options
&OPTIONS ASM
FILEONE ADA A
RC=00 FILEONE ADA A1
&OPTIONS DEF
FILETWO ADA A
RC=00 FILETWO ADA A1
&OPTIONS BLTZ
>>> ERROR IN INPUT LIST COMMAND SYNTAX
```

VMICMS File Name

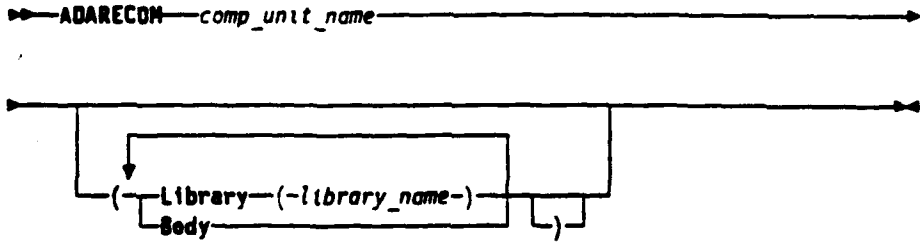
The name of the OUTPUT file takes the form *input_list_name* OUTPUT A, where *input_list_name* is the file name of the input list.

MVS Data Set Name

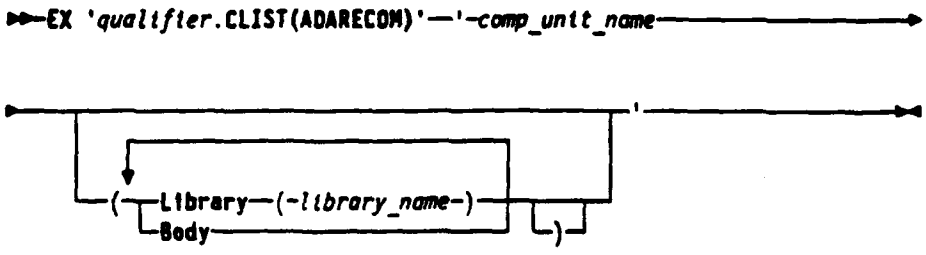
The name of the OUTPUT data set takes the form *qualifier*.OUTPUT(*input_list_name*), where *qualifier* is the current TSO logon identifier and *input_list_name* is the name of the input list.

Generating Recompilation Scripts

VMICMS Syntax



MVS Syntax



The ADARECOM command generates a recompilation order list. When a library consists of many compilation units that depend on (possibly multiple levels of) **with** statements, it can be difficult to determine the proper compilation order if

the specification of a unit needs to be recompiled. To use ADARECOM, you must have compiled the compilation units into a specified library at least once. ADARECOM reads the library to build a recompilation list of the units that depend upon the specified unit. This list contains the source file names of the units that need to be recompiled. Therefore, if you move an Ada compilation unit to a new source file, you must modify the recompilation list.

For ADARECOM to correctly generate the recompilation list, the association between file names and compilation units cannot change between the time they are compiled and the ADARECOM invocation.

The file list produced under either VM/CMS or MVS is suitable for input to the `INlist` option to the ADA command. For more on this subject, see "Compiling Multiple Source Files" on page 2-20.

Do not change the names of the source file, separate the specification and body into different files, or perform any other change that alters the relationship between that file and the compilation unit or units it contains between the time you make the recompilation list and the time you perform the recompilation. You can to edit the recompilation list after running ADARECOM, to make changes to the recompilation order list.

Also, ADARECOM produces a correct source file list only if each file contains a single compilation unit.

Precede the options by a blank space and a left parenthesis, and separate them from each other by blank spaces. A closing parenthesis is optional.

The `LIBrary` option allows you to specify the name of the library file, *library_name*, that ADARECOM is to read.

If you specify the `BODy` option, ADARECOM assumes that *comp_unit_name* refers to the body of a compilation unit. By default, *comp_unit_name* refers to the specification of a compilation unit.

Under VM/CMS
You can provide the *library_name* variable in either of two formats. The preferred format is *file_name file_type file_mode*. The other format is *file_mode:file_name.file_type*. In both formats, if you specify only the file name, *file_type* defaults to LIBRARY and *file mode* defaults to "*". If you do not specify a library file, ADA searches for ADA LIBRARY *.

The recompilation list goes into a file called *comp_unit* INLIST A.

The command

```
ADARECOM MY_PROG (L(MYLIB.LIBRARY))
```

generates a recompilation list that includes all units within MYLIB LIBRARY that must be recompiled if you recompile MY_PROG, also found in that library.

Under MVS
If you do not specify *lib_name*, the default is *qualifier*.ADA.LIBRARY.

The recompilation list goes into a file called *qualifier*.INLIST(*comp_unit*), where *qualifier* is your TSO logon identifier.

The command

```
EX 'ADA.CLIST(ADARECOM)' 'MY_PROG (L(MYLIB))'
```

generates a recompilation list that includes all units within *qualifier*.MYLIB that must be recompiled if you recompile MY_PROG, also found in that library

Separate Compilation of Generics

IBM Ada/370 supports separately-compiled generics. You can compile a generic specification in in file and its generic body in a separate file.

If you plan to compile your generics separately, compile the generic body before you attempt to instantiate the generic. The generic body must be compiled and visible in the library before the instantiation can occur.

You can *compile instantiations before* you compile the generic body. If you do, the compiler issues a warning. When you compile a generic body, all instantiations of the generic become obsolete. You must recompile all instantiations of the generic after you compile a new generic body.

For more information on the compiling and instantiation of generic units, see Chapter 12 of the LRM.

Chapter 3. Binding and Linking an Ada Program

The purpose of this chapter is to show you the different ways to invoke the IBM Ada/370 binder or the linkage editor, and to explain when you need to use them.

The flexibility in symbol naming allowed by Ada makes implementation of the language difficult in environments using pre-existing linkage editors and loaders. To alleviate this problem, IBM provides a special Ada linker called the IBM Ada/370 binder. It combines object modules produced by the IBM Ada/370 compiler and outputs them as a standard IBM object module.

This partially-linked object module is further processed by the system linker/loader to produce an executable load module. The IBM Ada/370 binder provides full support of Ada requirements for symbol naming. It also drastically reduces the number of external definitions and references that must be processed by the host system linker.

An Ada program can use pragma Interface to call subprograms written in a programming language other than Ada. The system linker puts the standard-format object modules produced for these subprograms into the executable load module it creates for an Ada program.

The IBM Ada/370 binder also includes run-time environment routines as part of its output.

Using the IBM Ada/370 Binder

To invoke the IBM Ada/370 binder, compile an Ada main program using the `MAIn` option of the ADA command. The system invokes the binder. The binder can produce a link map describing the contents of the partially linked object module it generates. The link map provides you with detailed information about the run-time memory locations of the various pieces of code that make up your program.

Another option to the ADA command, `Bind`, causes IBM Ada/370 to bypass the compilation step. This allows you to bind a compilation unit that you have previously compiled as a library unit as a main program.

The `Generate` option of the ADA command takes binder output and uses system utilities to generate a load module.

For more information on the `MAIn`, `Bind`, and `Generate` options to the ADA command, see "The Compiler Options" on page 2-6.

You must rebind your main program when you recompile any Ada compilation units used in the program. You do not have to rebind the program if you recompile non-Ada routines that your Ada program calls, but you still have to link the program again with the linkage editor or loader.

Invoking the Binder with Job Control Language (JCL)

This section describes how to invoke the binder as a batch job under MVS using Job Control Language (JCL). For information on how to invoke the compiler using JCL, see "Compiling a Program with Job Control Language (JCL)" on page 2-3.

The ADAB cataloged procedure invokes the IBM Ada/370 binder to bind an Ada main program that has been compiled using the IBM Ada/370 compiler. The output of the binder is an System/370 relocatable object data set. You can submit this data set to the linkage editor to generate an executable load module.

```
//MYPROG    JOB ,'  ',MSGCLASS=D,MSGLEVEL=(1,1),NOTIFY=USER1,
//        CLASS=A
//*
//*  PURPOSE:  TO RUN THE ADA BINDER
//*
//BIND    EXEC PROC=ADAB,UNIT=HELLO,
//         USER=USER1
```

Figure 3-1. Using the ADAB Cataloged Procedure to Invoke the Binder

The preceding example shows a job, called MYPROG, which binds the Ada main compilation unit HELLO. The user identifier USER1 is specified with the USER variable. This variable is used as a high-level qualifier to construct data-set names for the compiler, such as USER1.ADA.LIBRARY. This library is the default Ada library. Your job card will probably be different, because it depends on your site's conventions.

As this job executes, the compiler creates relocatable object code in USER1.OBJ(HELLO). This object code was generated for the Ada main compilation unit called HELLO.

A sample of the ADAB cataloged procedure appears in Figure 3-2. The exact location of ADAB may depend on your site's conventions.

```
//ADAB     PROC BNDPRM=' ',MEMSIZE=8196K,
//          STPLIB='ADA110.LOADLIB',MAXTIME=60,
//          VIO=VIO,SYSDA=SYSALLDA,SYSOUT='*',UNIT=' '
//*****************************************************************
//*                        INVOKE THE BINDER
//*****************************************************************
//STEP1    EXEC PGM=EVGBIND,PARM='&UNIT. ( &BNDPRM',REGION=&MEMSIZE,
//           TIME=&MAXTIME,DYNAMNBR=65,COND=(4,LT)
//STEPLIB   DD  DSN=&STPLIB,DISP=SHR
//CONOUT    DD  SYSOUT=&SYSOUT,DCB=(LRECL=120,BLKSIZE=120)
//ADALIB    DD  DSN=&USER..ADA.LIBRARY,DISP=SHR
//ADAOBJ    DD  DSN=&USER..OBJ,DISP=(MOD,CATLG,CATLG),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120,DSORG=PO,BUFNO=4),
//          SPACE=(80,(16000,16000,20)),UNIT=&SYSDA
//ADAMAP    DD  DSN=&USER..ADAMAP,DISP=(MOD,CATLG,CATLG),
//          DCB=(RECFM=VB,LRECL=1023,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//          SPACE=(132,(1000,2000,20)),UNIT=&SYSDA
//ADADMAP   DD  DSN=&USER..DEBUGMAP,DISP=(MOD,CATLG,CATLG),
//          DCB=(RECFM=VB,LRECL=1023,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//          SPACE=(132,(1000,2000,20)),UNIT=&SYSDA
//ADALIST   DD  DSN=&USER..LISTING,DISP=(MOD,CATLG,CATLG),
//          DCB=(RECFM=VBA,LRECL=259,BLKSIZE=3120,DSORG=PO,BUFNO=2),
//          SPACE=(132,(500,2000,20)),UNIT=&SYSDA
//ADAUT1    DD  SPACE=(132,(500,2000)),
//          DCB=(RECFM=FB,LRECL=136,BLKSIZE=3400,DSORG=DA,BUFNO=2),
//          UNIT=&SYSDA
//ADAUT2    DD  SPACE=(132,(500,2000)),
//          DCB=(RECFM=VB,LRECL=136,BLKSIZE=3120,DSORG=PS,BUFNO=2),
//          UNIT=&VIO
//ADAUT3    DD  SPACE=(132,(500,2000)),
//          DCB=(RECFM=FB,LRECL=1028,BLKSIZE=2056,DSORG=DA,BUFNO=2),
//          UNIT=&SYSDA
//ADAUT4    DD  SPACE=(132,(500,2000)),
//          DCB=(RECFM=FB,LRECL=132,BLKSIZE=2640,DSORG=DA,BUFNO=2),
//          UNIT=&SYSDA
```

Figure 3-2. ADAB Cataloged Procedure

Symbolic Variables for ADAB Cataloged Procedure

The ADAB cataloged procedure includes several symbolic JCL substitution variables you can modify to specify the various options available.

Symbolic Variable	Description
BNDPRM	Specifies options to the binder in the PARM field. A list of options you can specify in BNDPRM, along with a syntax diagram, appears in "Binder Options for Use with ADAB Cataloged Procedure" on page 3-5.
MAXTIME	Sets a maximum amount of time for the binder job step to run (using the TIME parameter on a JCL EXEC card). The default value is 60 minutes, but can be changed when the cataloged procedure is installed.
MEMSIZE	Specifies the amount of memory in which to run the binder job step (using the REGION option on a JCL EXEC card). The greater the memory, the better the binder's performance. The default value is 8196Kb, but this value may be changed when you install the cataloged procedure.
STPLIB	Indicates the data-set name of the partitioned data set that contains the binder module, MVSBINDE. The default is the load library ADA.LOADLIB but you can change this default when you install the cataloged procedure.
SYSDA	Specifies the UNIT for permanent data set allocations. The default is SYSALLDA, but this name can be changed depending on your site's conventions.
SYSOUT	Identifies the output class for the binder output. The default is "*", but this default can be changed, depending on your site's conventions.
UNIT	Indicates the compilation unit to be bound. You must specify this variable.
USER	Indicates the high-level qualifier level qualifier required to build data set names used by the binder. You must specify this variable. It is common to set it to your TSO logon identifier.
VIO	Specifies the UNIT for temporary data set allocations. The default is VIO, but this default can be changed depending on your site's conventions.

Binder Options for Use with ADAB Cataloged Procedure

This section defines the standard options for the BNDPRM symbolic substitution variable. This variable, found in STEP2 of the ADAB cataloged procedure, specifies options to the binder in the PARM field.

The binder options have the following syntax:

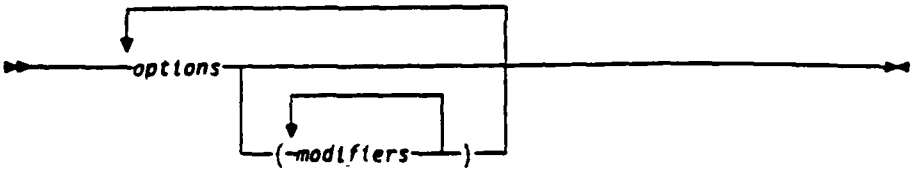


Table 3-1 lists the valid options, their modifiers, and defaults. Descriptions of the options appear on the pages shown.

Table 3-1 Binder Options for JCL

Option	Default	Function	Page
ASM [NOGEN | NOSYS]	NOASM	Assembly listing. Suppress listing of expanded generics. Suppress listing of system-supplied generics.	2-9
DDNAMES *old_name=new_name*		Specify the Data Description (DD) names that identify the data sets used by the compiler and binder. It must include at least one modifier.	2-12
DEBUG	NODEBUG	Output information for debugging.	2-13
MAP	NONAP	Produce linkage map during binding.	2-17
TRACE	NOTRACE	Display diagnostic messages from the compiler. Only for use in submitting problems to IBM.	2-19

Linking Programs that Call Non-Ada Routines

Pragma Interface enables Ada compilation units to call non-Ada routines. The following sections show you how to compile and link programs that take advantage of this feature.

User-written routines do not reside in the Ada library system. The only method available to connect non-Ada routines with the Ada routines that call them is to load them under VM/CMS or link-edit them under MVS A call to a non-Ada routine results in the generation of an external reference. This external reference is unresolved following normal ADA processing. You must take special steps in order to resolve virtual address constants to non-Ada routines.

Under VM/CMS

The following table explains the data sets used in calling non-Ada routines.

Table 3-2. VM/CMS Files Used in Calling Non-Ada Routines

File	Description
TEST ADA A	File containing source for the main program.
TEST	Compilation unit name.
TEST TEXT A	File containing the object code of the Ada routines.
ROUTINE TEXT A	File containing the object code of the non-Ada routines.

The following example shows the correct procedure for compiling and linking

```
ADA TEST (MAIN NORUN
LOAD TEST ROUTINES
GENMOD TEST
```

The ADA command creates the object file TEST TEXT A. The LOAD loads the object files TEST TEXT and ROUTINES TEXT into virtual storage and establishes the proper linkages between them. The order in the LOAD is important. The Ada module must go first. The GENMOD command uses the two object files to create a load module with the name TEST MODULE A.

You may need to precede the commands in this example with a GLOBAL TXTLIB command to resolve any missing external references from the LOAD command. The need for its use depends on how you load the non-Ada routines. For more information on the GLOBAL command, see the *Virtual Machine/System Product CMS Command and Macro Reference*.

Under MVS

Non-Ada routines do not reside in the Ada library system. The only method available to connect non-Ada routines with the Ada routines that call them is to link-edit them. A call to a non-Ada routine results in the generation of an external reference. This external reference is unresolved following normal ADA processing. You must take special steps in order to resolve virtual address constants to non-Ada routines. This section explains those steps.

The following table explains the data sets used in calling non-Ada routines.

Table 3-3 (Page 1 of 2). MVS Data Sets Used in Calling Non-Ada Routines

Data Set	Description
qualifier.TEST.ADA	Data set containing source for the main program, whose compilation unit name is Test.
qualifier.NONADA.OBJ	Data set containing the object code of the non-Ada routines.
qualifier.OBJ	Partitioned data set containing the object code of the Ada routines.
qualifier LOAD	Partitioned data set containing the executable load modules; also called the "load library."

There are two methods for compiling and linking an Ada program with non-Ada routines. The first involves binding with NOGenerate, then linking the foreign

language code manually with the linkage editor. The second involves placing the code in a partitioned data set (PDS) such that the object goes into the object library created by the Bind option. The following two examples show how to use these methods.

Using the Linkage Editor

First compile the main program, using the NOGenerate option.

```
EX 'ADA.CLIST(ADA)' 'TEST.ADA (MAIN NOGENERATE'
```

This member contains one or more unresolved references to non-Ada code. The following call to the linkage editor resolves the unresolved external references associated with those calls.

```
LINK ('USER1.NONADA.OBJ','USER1.OBJ(TEST)')
```

The TEST load library member is now fully linked and ready to execute.

Using a Partitioned Data Set

If you choose to use a partitioned data set, use the following steps:

1. Place the foreign language routine's object code into a partitioned data set.
2. Issue the TSO ALLOC command for a DD name of SYSLIB and then associate this with the PDS containing the non-Ada object code.

   ```
   ALLOC DD(SYSLIB) DA('USER1.NONADA.OBJ') SHR
   ```

3. Bind the main program (or compile and bind) without using the NOGenerate option. The LINK within ADA will refer to the SYSLIB allocation as it attempts to resolve references to the non-Ada routines.

The TEST load library member is now fully linked to be executed.

For more information on LINK, see the LINK command in the IBM publication, *MVS/Extended Architecture* TSO *Extensions* TSO *Command Language Reference*.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
    ..........

package STANDARD is

   type INTEGER is range -2_147_483_648 .. 2_147_483_647;
   type SHORT_INTEGER is range -32_768 .. 32_767;

   type FLOAT is digits 6 range -7.23701E+75 .. 7.23701E+75;
   type LONG_FLOAT is digits 15 range -7.23700557733225E+75
                                .. 7.23700557733225E+75;

   type DURATION is delta 2#1.0#E-14 range -86400.0 .. 86400.0;

end STANDARD;

    ..........
```

ATTACHMENT A

APPENDIX F
OF THE LANGUAGE REFERENCE MANUAL

The Ada language definition allows for certain target dependencies in a controlled manner. This section, called Appendix F as prescribed in the LRM, describes implementation-dependent characteristics of the IBM Ada/370, Version 1.1.0 running under CMS or MVS.

1. Implementation-Defined Pragmas

PRAGMA INTERFACE(Assembly, <subroutine_name>);

PRAGMA INTERFACE(Assembler, <subroutine_name>);

PRAGMA INTERFACE(Fortran, <subroutine_name>);

PRAGMA SUPPRESS_ALL;

to cause Pragma SUPPRESS to be invoked simultaneously for all the following condition_names: access_check, discriminant_check, index_check, length_check, division_check, elaboration_check, and storage_check.

PRAGMA NO_SUPPRESS (<identifier>);

to prevent the suppression of checks within a particular scope. Particularly useful when a section of code that relies upon predefined checks executes correctly, but, for performance reasons, the suppression of checks in the rest of the code is needed.

PRAGMA COMMENT (string_literal);

embeds string_literal into object code.

PRAGMA IMAGES (enumeration_type, <immediate>|<deferred>);

generates a table of images for the enumeration type. **deferred** causes the table to be generated only if the enumeration type is used in a compilation unit.

PRAGMA INTERFACE_INFORMATION

(<name>,
<link_name>,
<mechanism>,
<parameters>,
<clobbered_regs>);

when used in association with pragma INTERFACE, will provide access to any routine whose name can be specified by an Ada string literal.

PRAGMA PRESERVE_LAYOUT (ON => <Record_Type_Name>);

forces the compiler to maintain the Ada source order of components of a given record type, thereby preventing the compiler from performing this record layout optimization.

*PRAGMA INLINE (procedure_name);

to specify a subprogram body which should be expanded inline at each call whenever possible.

**PRAGMA OS_TASK (priority);

to specify the relative urgency of each MVS task created.

**PRAGMA ALLOCATION_DATA

(<access_type>,
<residence_mode>,
<allocation_duration>,
<subpool_number>,
<discrete_user_data>);

to associate MVS virtual storage attributes with an Ada access type.

*Note that PRAGMA INLINE is effective only when the optimizing option is selected at compile time. If optimizing is not selected the pragma is ignored and a warning is issued.

**Note that PRAGMA OS_TASK and PRAGMA ALLOCATION_DATA are effective only when compiling for an MVS target. Both pragmas require that an MVS runtime be present.

2. Implementation-Defined Attributes

2.1. Integer Type Attributes

Extended_Image (Item, <Width>, <Base>, <Based>, <Space_IF_Positive>);

to return the image associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that the value of Item is an integer literal with no underlines, no exponent, no leading zeroes (but a single zero for the zero value), and a minus sign if negative.

Extended_Value (Item);

to return the value associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that given a string, it reads an integer value from the beginning of the string. The value returned corresponds to the sequence input.

Extended_Width (<Base>, <Based>, <Space_IF_Positive>);

to return the width for a subtype specified.

2.2. Enumeration Type Attributes

Extended_Image (Item, <Width>, <Uppercase>);

to return the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that given an enumeration literal, it will output the value of the enumeration literal (either an identifier or a character literal). The character case parameter is ignored for character literals.

Extended_Value (Item);

to return the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that it reads an enumeration value from the beginning of the given string and returns the value of the enumeration literal that corresponds to the sequence input.

Extended_Width;

to return the width for a specified subtype.

2.3. Floating Point Attributes

Extended_Image (Item, <Fore>, <Aft>, <Exp>, <Base>, <Based>);

to return the image associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part.

Extended_Value (Item);

to return the value associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it skips any leading zeroes, then reads a plus or minus sign if present, then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input.

Extended_Digits (<Base>);

to return the number of digits using base in the mantissa of model numbers of the specified subtype.

2.4. Fixed Point Attributes

Extended_Image (Item, <Fore>, <Aft>, <Exp>, <Base>, <Based>);

to return the image associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part.

Extended_Value (Image);

to return the value associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it skips any leading zeroes, reads a plus or minus sign if present, then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input.

Extended_Fore (<Base>, <Based>);

to return the minimum number of characters required for the integer part of the based representation specified.

Extended_Aft (<Base>, <Based>);

to return the minimum number of characters required for the fractional part of the based representation specified.

3. Package SYSTEM

The current specification of package SYSTEM is provided below.

```
With Unchecked_Conversion;

PACKAGE System IS
```

```
--=============================================================
-- CUSTOMIZABLE VALUES
--=============================================================

TYPE Name    IS (MC68000, ANUYK44, IBM370);

System_Name  : CONSTANT name := IBM370;

Memory_Size  : CONSTANT := (2 ** 24)-1;
Tick         : CONSTANT := 1.0 / (10 ** 6);

--=============================================================
-- NON-CUSTOMIZABLE, IMPLEMENTATION-DEPENDENT VALUES
--=============================================================

Storage_Unit : CONSTANT := 8;
Min_Int      : CONSTANT := -(2 ** 31);
Max_Int      : CONSTANT := (2 ** 31) - 1;
Max_Digits   : CONSTANT := 15;
Max_Mantissa : CONSTANT := 31;
Fine_Delta   : CONSTANT := 1.0 / (2 ** Max_Mantissa );

Subtype Priority IS Integer RANGE 0 .. 255;

--=============================================================
-- ADDRESS TYPE SUPPORT
--=============================================================

type Memory is private;
type Address is access Memory;

Null_Address : Constant Address := null:

type Address_Value is RANGE -(2**31) .. (2**31)-1;

Hex_80000000 : constant Address_Value := - 16#80000000#:
Hex_90000000 : constant Address_Value := - 16#70000000#:
Hex_A0000000 : constant Address_Value := - 16#60000000#;
Hex_B0000000 : constant Address_Value := - 16#50000000#;
Hex_C0000000 : constant Address_Value := - 16#40000000#;
Hex_D0000000 : constant Address_Value := - 16#30000000#;
Hex_E0000000 : constant Address_Value := - 16#20000000#;
Hex_F0000000 : constant Address_Value := - 16#10000000#;

function Location is new Unchecked_Conversion (Address_Value, Address);

function Label (Name: String) return Address:
pragma Interface (META, Label);
```

```
--------------------
-- CALL SUPPORT
--------------------

type Subprogram_Value IS
record
  Proc_addr       : Address;
  Parent_frame    : Address;
end record;

Max_Object_Size    : CONSTANT := Max_Int;
Max_Record_Count   : CONSTANT := Max_Int;
Max_Text_Io_Count  : CONSTANT := Max_Int-1;
Max_Text_Io_Field  : CONSTANT := 1000;

private
 type Memory is
 record
  null;
 end record;

end SYSTEM;
```

4. Representation Clauses

This implementation supports address, length, enumeration, and record representation clauses with the following exceptions:

> Address clauses are not supported for package, for entry, for tasktype, for subprograms.
>
> Enumeration clauses are not supported for boolean representation clauses.

The size in bits of representation specified records is rounded up to the next highest multiple of 8, meaning that the object of a representation specified record with 25 bits will actually occupy 32 bits.

Non-supported clauses are rejected at compile time.

5. Implementation-Generated Names

There are no implementation-generated names denoting implementation-dependent components. Names generated by the compiler shall not interfere with programmer-defined names.

6. Address Clause Expression Interpretation

Expressions that appear in Address clauses are interpreted as virtual memory addresses.

7. Unchecked Conversion Restrictions

Unchecked_Conversion is allowed except when the target data subtype is an unconstrained array or record type. If the size of the source and target are static and equal, the compiler will perform a bitwise copy of data from the source object to the target object.

Where the sizes of source and target differ, the following rules will apply:

- If the size of the source is greater than the size of the target, the high address bits will be truncated in the conversion.
- If the size of the source is less than the size of the target, the source will be moved into the low address bits of the target.

The compiler will issue a warning when Unchecked_Conversion is instantiated with unequal sizes for source and target subtype. Unchecked_Conversion between objects of different or non-static sizes will usually produce less efficient code and should be avoided, if possible.

8. Implementation-Dependent Characteristics of the I/O Packages

- Sequential_IO, Direct_IO, and Text_IO are supported.
- Low_Level_IO is not supported.
- Unconstrained array types and unconstrained types with discriminants may not be instantiated for I/O.
- File names follow the conventions and restrictions of the target operating system.
- In Text_IO, the type Field is defined as follows: subtype Field is integer range 0..1000;
- In Text_IO, the type Count is defined as follows: type Count is range 0..2_147_483_646;